Modeling Interfaces and Interface Protocols

Architecture verification for SMAP brought IV&V face to face with Model Driven approach to Architecture Definition

Services offered at a SW interface specified with Protocols

IV&V Workshop 2010
Karl Frank
TASC General Scientist

Topics

Architecture Verification with SW Interface and Component Concepts

- Software Application Architecture: Components and Interfaces
  - OO and SOA concepts differentiated from others
  - Components, Interfaces and Service Protocols as encountered in SMAP
- How these are modeled in UML 2
- How such models can be used in Architecture Verification
- Examples: Manual Transmission and Space Camera

- What you may learn:
  - Architecture verification for a model-driven project whose architecture is based on OO and SOA concepts
  - Details:
    - protocol statemachine contrasted with behavioral statemachine
    - How to tell from protocol statemachine, what service invocation sequences to test for, which should work – and which probably won't
    - Relevance to commercialization and reuse of COTS components

Motivation

- Keeping up: NASA projects using protocol statemachines
  - IV&V verifying architecture, design, and test plans for SMAP, a project that is defines its architecture to us in these terms
- Architecture modeling with application level components and interfaces supports tracing from high level down to tests, and back up, for projects that model this way
  - Interfaces an intermediate stage: services not how they are realized
  - Technique for verifying that a service interface is adequately specified
- Fixing hole in command language specs:
  - fallacy of the lexicon
    - No language is defined by its lexicon
    - Protocol statemachines fill one gap by defining validity in terms of context
- Mission incident could have been prevented by this approach

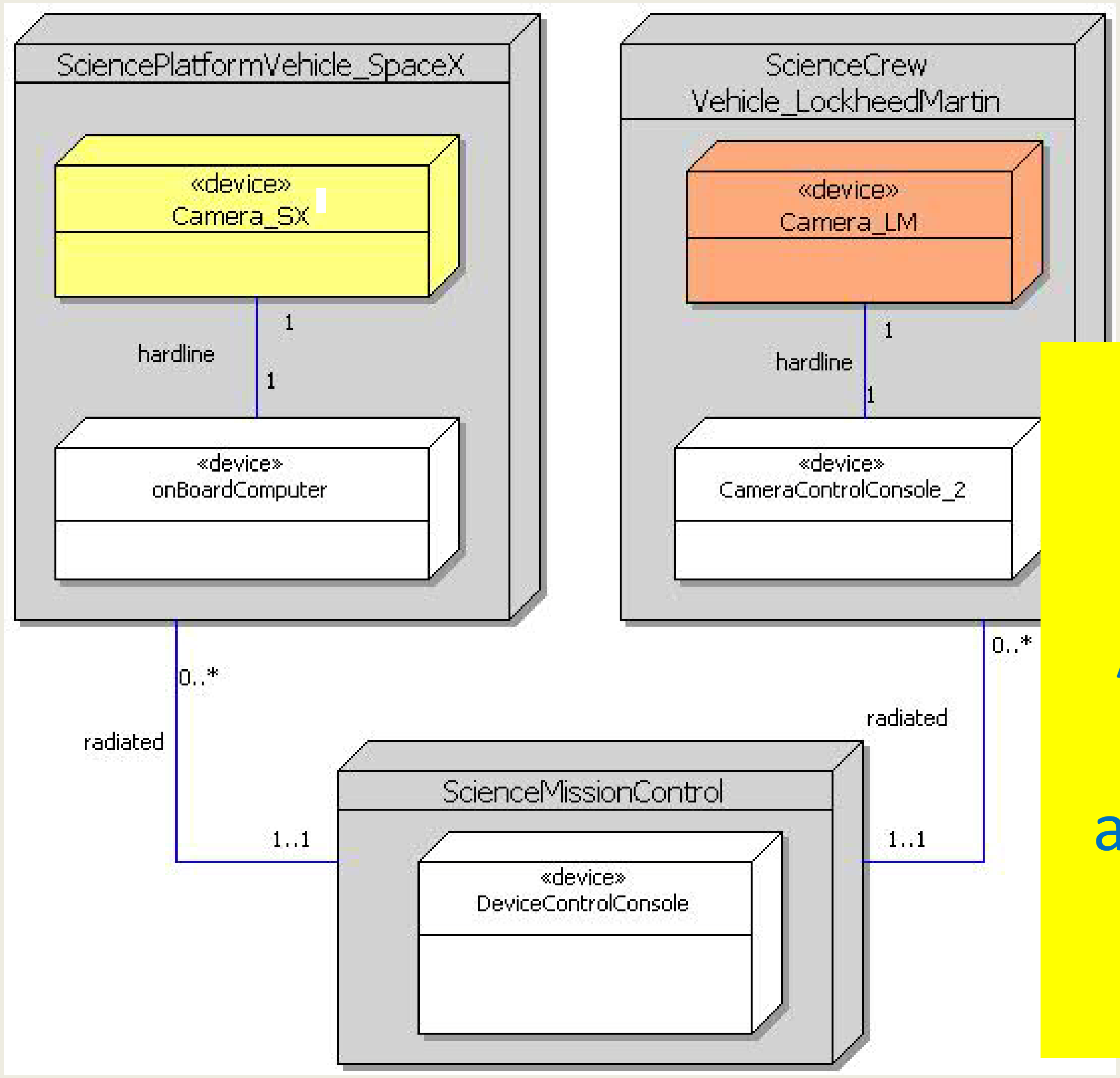
Failure to acquire fly-by photos

- A science mission had on-board command-operated camera, to acquire images, to be beamed back
- At certain points in the mission, `TakePicture` commands were sent
  - Syntax checking at mission control and on-board confirmed the commands were well formed
  - Outcome could not be seen by scientists until the opportunity for getting desired images was past
  - After anxious waiting, radiated datastream that should have had the images came back, empty!
- `TakePicture` needed to be preceded by an `EnableCamera` command to have intended effect

Ongoing Fictional Example

- Next slides show deployment and component diagrams of a possible system architecture, using replaceable components on a variety of vehicles
- Example is fictionalized.
- Start with physical and comm models, move on to a App level component and interface model
- From there to Protocol Statemachine
  - A specialized form of statemachine for defining rules client needs to follow in using the services
  - Here, to use camera services, regardless of vendor

Example: Deployment Diagram



Physical context, not a software-centric view. Application level SW Architecture and SW Interfaces need a different approach

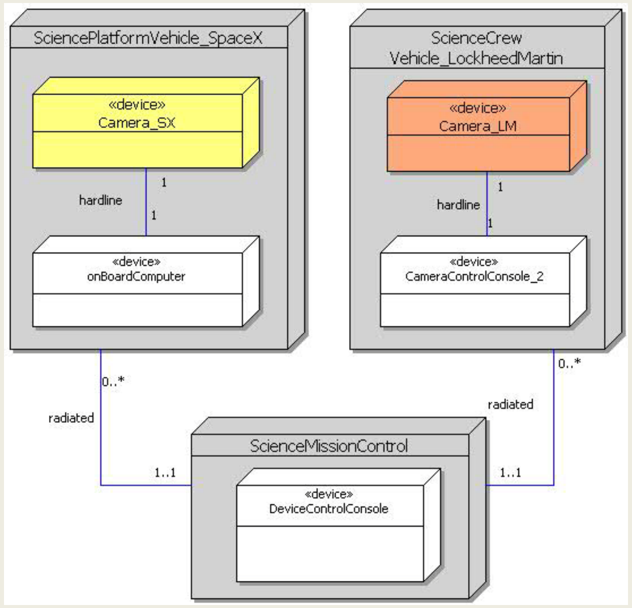
Interfaces & Services modeled as abstractions to be realized later

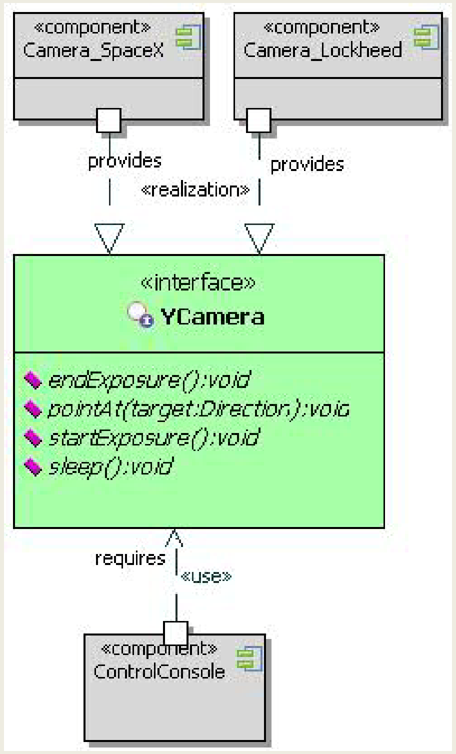
- These interfaces are NOT separate executables mediating between clients and suppliers
  - May be realized by “wrappers”, ORB brokers, etc.
- Recognizes big differences from physical interfaces
  - Communicating system health and status on a data stream is like fuel flowing thru pipes, but the wrong paradigm for SOA – which is more like a remote procedure call
    - because Services are invoked from client side.
    - delivery of the service often is not a flow back to the client, but the performance of a local behavior
  - SMAP defines its architecture following the OO paradigm on which UML 2 based its notation.

Real Building Blocks of Software Architecture are components, CSCIs

- Software interfaces exposed by SW components at ports, where other components can invoke services available at that interface
  - Service request originated by the Client
  - May ride atop an ongoing data stream, which does fit a flow paradigm in communication engineering
- App Services reside in the top (application) layer of communication model introduced by OSI

From SW IV&V perspective



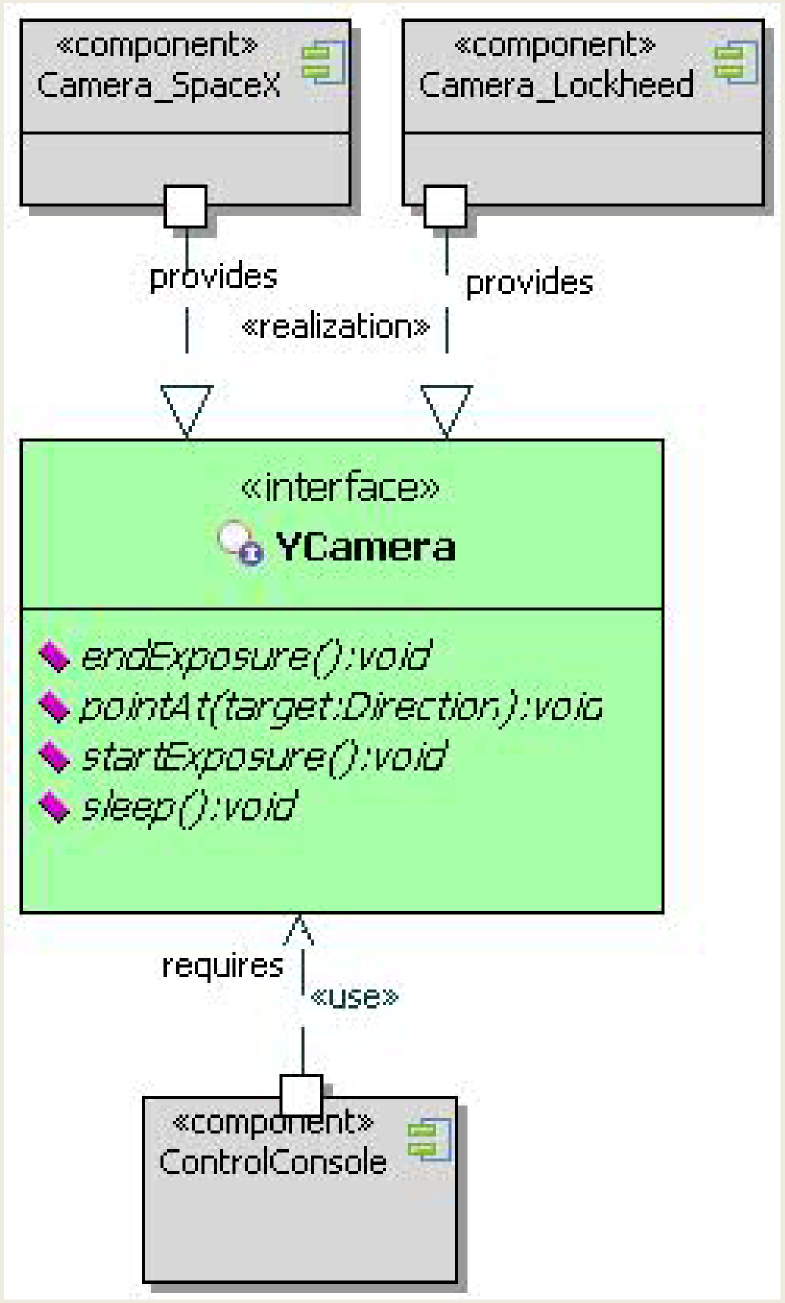


- Systems and Communication Engineering concerns are not our topic here
- Application architecture model of SMAP hides those concerns

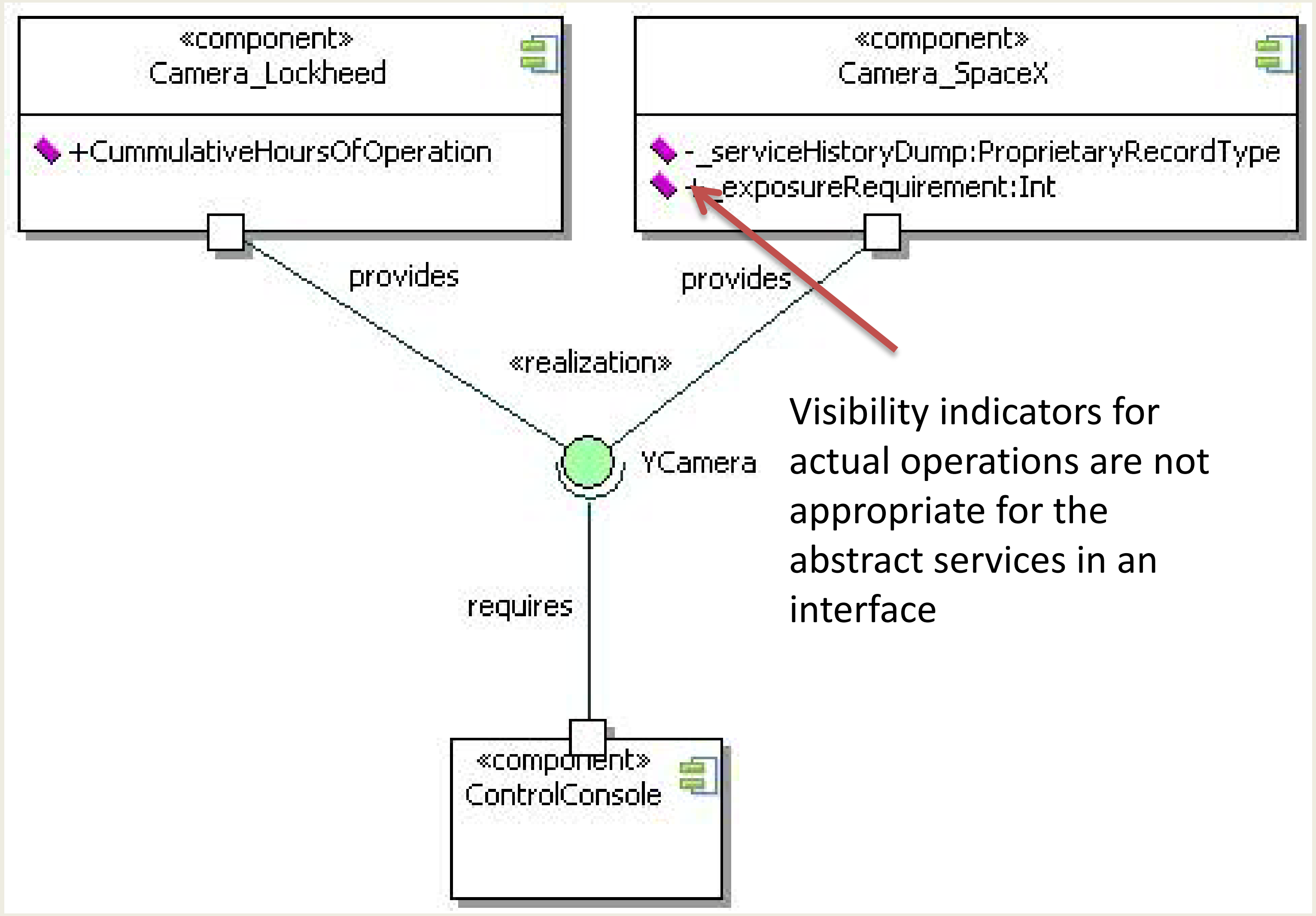
Application Layer rides on top of physical systems and communication architecture

2 Components realize same interface used by 1 client

is a central concern in validating an application SW architecture based on OO paradigm

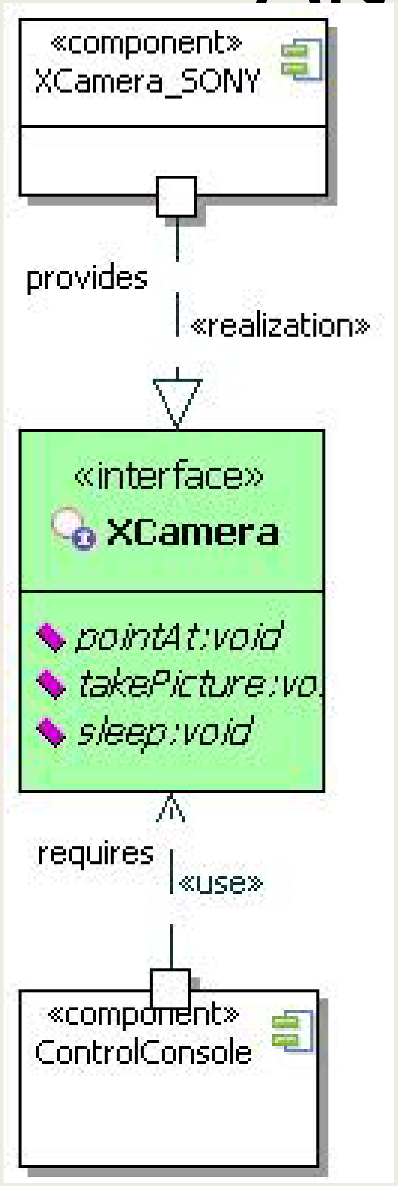


Internal differences among cameras likely, best kept private



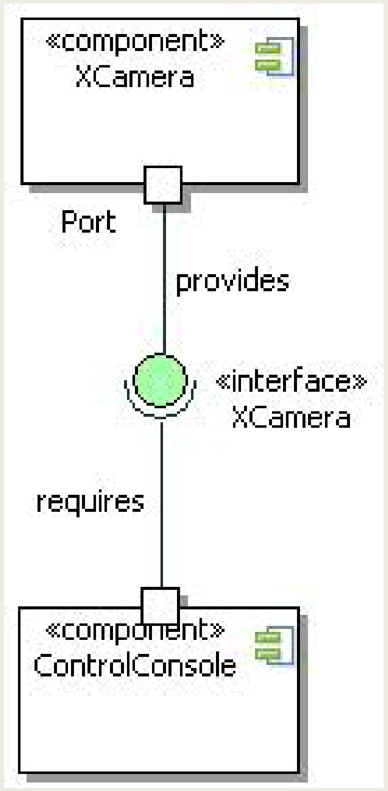
Visibility indicators for actual operations are not appropriate for the abstract services in an interface

Alternative Views



- Ball and Socket view hides discrete services

Alternative terminology and notation for Ball & Socket … to show services offered at the interface



SMAP Architecture Model Concepts

- *Interface* accessed thru *Port* on *Component*
  - Models components as black boxes to maintain independence of IV&V models from implementation
  - *Port* typed for static check of data in or out
- *Interface* is an abstraction:
  - user does not need to know about implementation,
  - Protocol encompasses all services offered at interface
- Hence, topic is really a unity:
  - Modeling Component Interfaces without modeling protocol only establishes the static correctness of a component architecture.
  - Topic goes beyond static architecture audit to dynamic testing of architecture thru its interfaces

Component contrasted with Interface

- Many actual components from diverse developers realize the SAME interface
  - Assembling a valid system depends on interfaces
    - Interchangeability of components depends on equivalence of interfaces
  - Interfaces define external black box view
  - Interfaces declare services as ABSTRACT operations
  - There are rules for “correct” use of an interface
    - Recall the example of the manual transmission
    - We want a way to define a dynamic black box view

Model services offered at interface

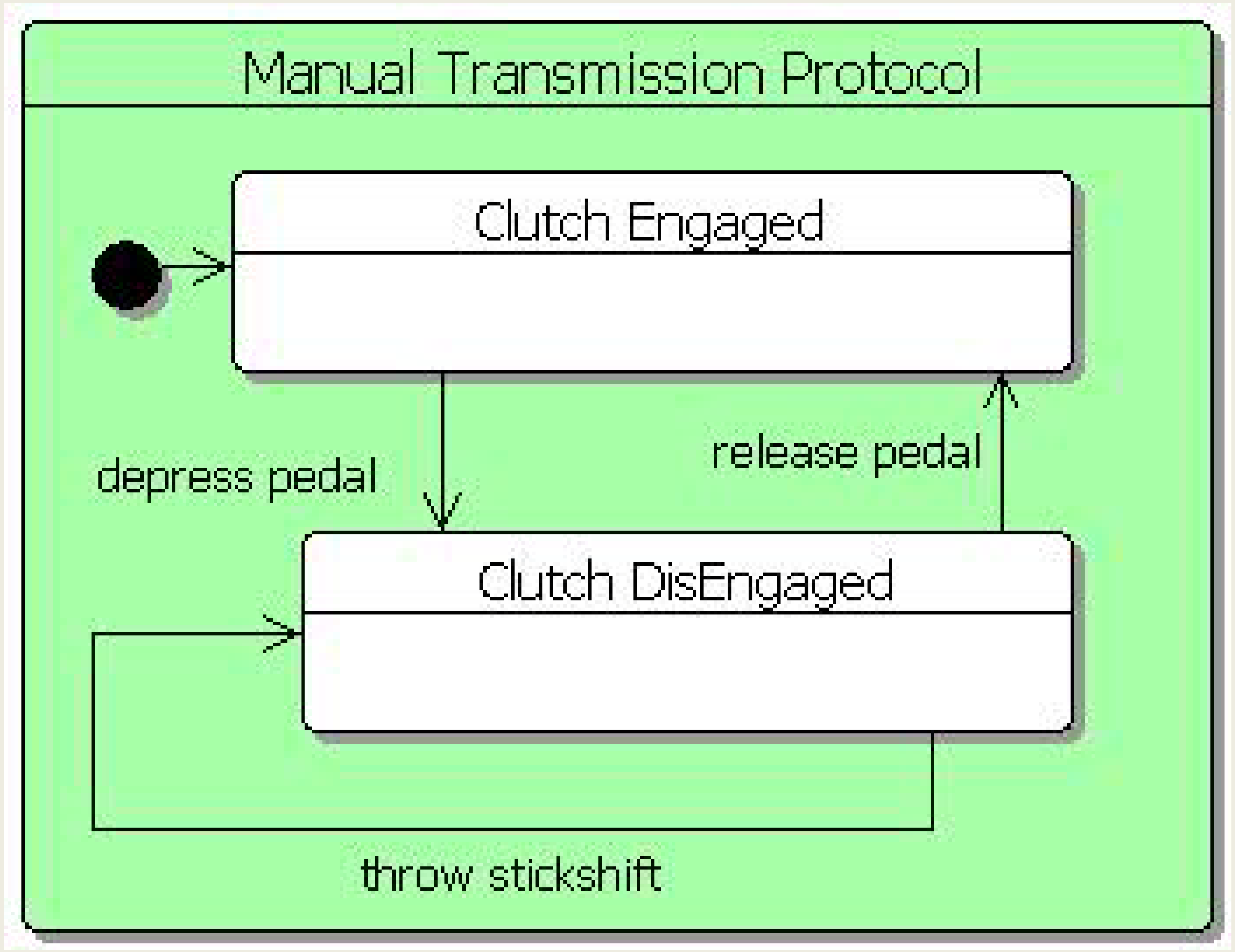
- By contract: preconditions, invariants, postconditions for each service, one – at – a time
  - This is a *static* representation, method signature plus
  - Services offered in an interface often part of a set, used in certain dynamic contexts, not others
  - Not modeled as <u>Behavior</u> (no <u>actions</u> represented)
- Need an approach for dynamic context modeling
- Consider dynamics of interface for operating a car:
  - Don’t move the stickshift and THEN step on clutch
  - You step on the clutch and THEN shift
  - We need a way to model these contextual rules

Use a Protocol Statemachine

- Assume user sends messages to car transmission
- User doesn’t need to know about what’s under the hood, Except for what operations are OK in what context
- Convention is: valid messages are those that trigger transitions in a simplified statemachine

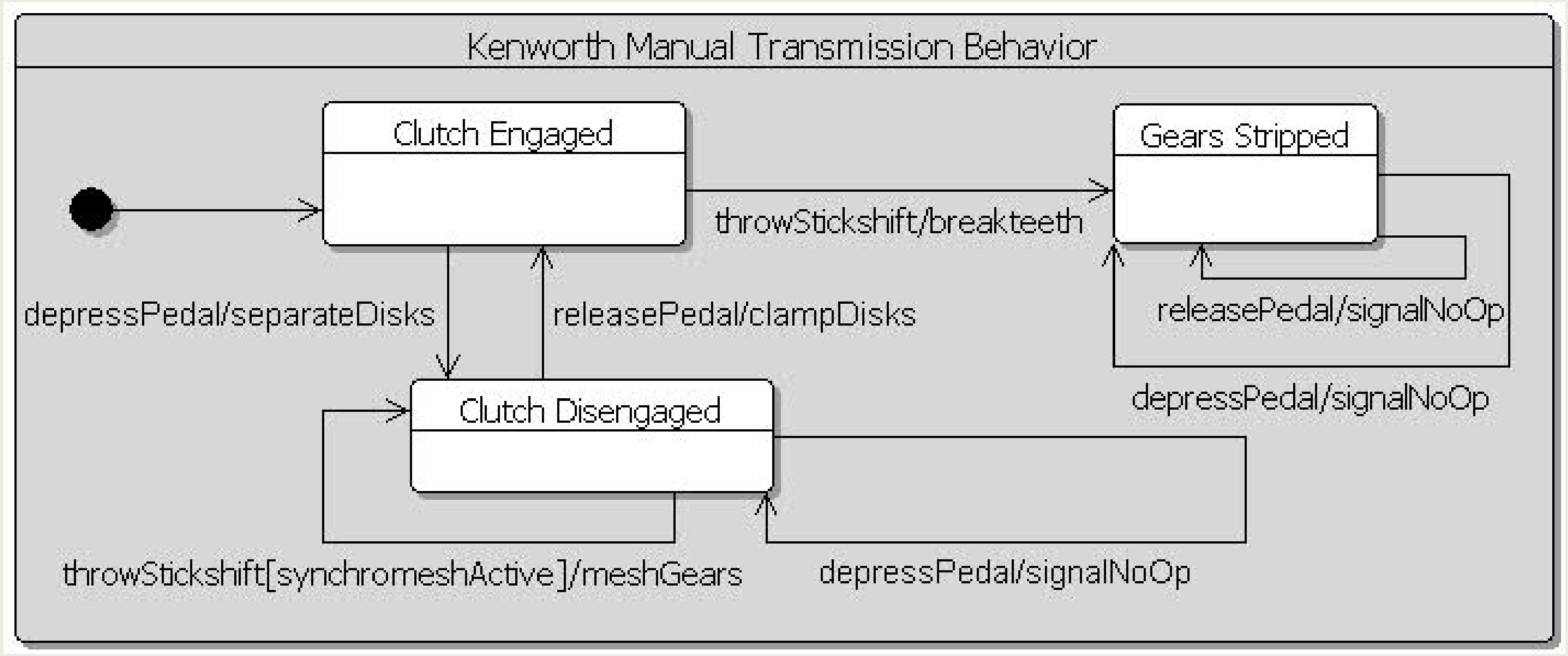
Manual Transmission as a black box operated by sending messages to its interface

Protocol for using manual transmission



Behavioral Statemachine for Actual Component

- Behavioral statemachine shows internals



Protocol statemachine

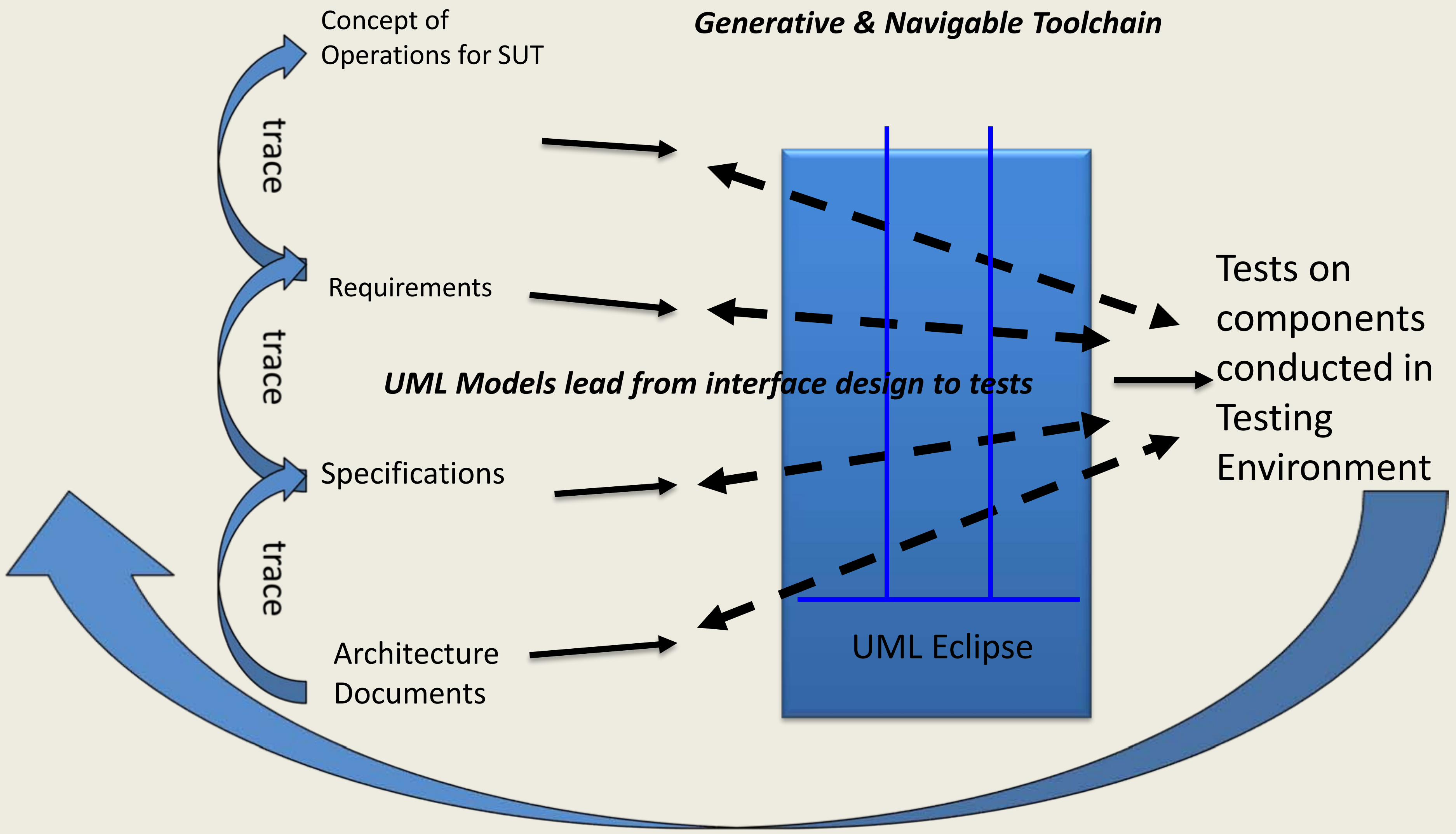
- Simpler than ordinary UML statemachines
- For characterizing the rules for invoking services at a software interface
  - Protocol statemachines are linked in UML models to Interfaces, whereas “ordinary” UML statemachines are linked to components which realize interfaces
  - Protocol statemachines are for defining the rules for using the services exposed at an interface, and so they conceal the actual workings of the component
    - Transitions are triggered by invocations of service – the messages that arrive at the interface
    - Shows changes in state externally visible (meaning, the modal behavior that matters to the client using the interface)
    - No effects (internal call to private objects) allowed transitions

From UML 2 Spec

Protocol state machines are used to express usage protocols. Protocol state machines express the legal transitions that a classifier can trigger. The state machine notation is a convenient way to define a lifecycle for objects, *or an order of the invocation of its operation.*

Protocol state machines do not preclude any specific behavioral implementation. They enforce legal usage scenarios. Interfaces and ports can be associated to this kind of state machines.

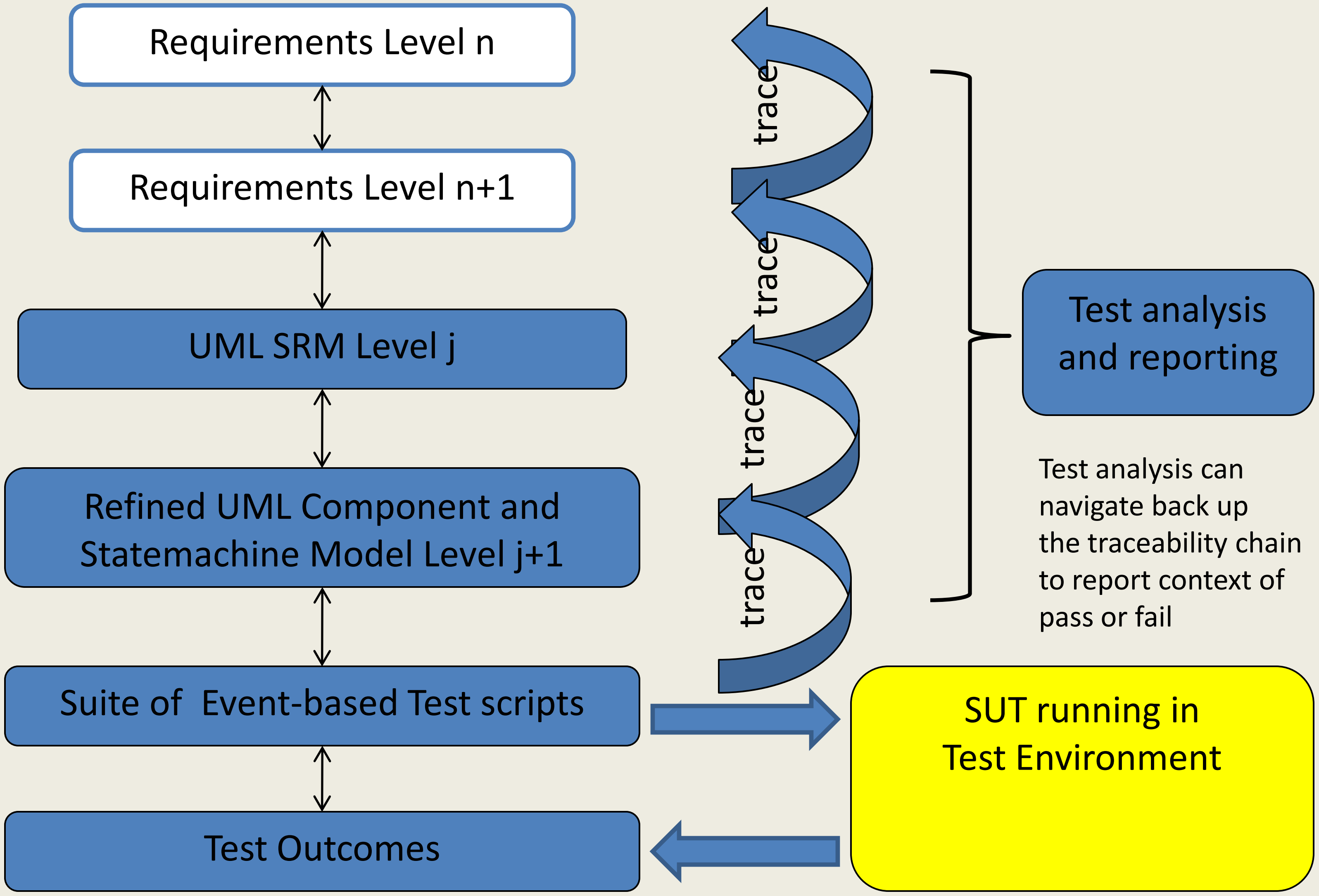
Conceptual Overview


Concept of Operations for SUT
Generative & Navigable Toolchain
trace
Requirements
trace
UML Models lead from interface design to tests
Specifications
trace
Architecture Documents
UML Eclipse
Tests on components conducted in Testing Environment

A Tool for Testing

- Objectives
  - Close a gap in our ability to define relevant tests
    - What system should not be expected to do
      - If test violates protocol, is the test is inappropriate?
      - Intent of a protocol statemachine as design artifact is distinctive
    - Adequate interface specifications for decoupled architectures
    - Sets state for later ability to verify conformance of implementations to interface specifications

Traceability Summary


Requirements Level n
Requirements Level n+1
UML SRM Level j
Refined UML Component and Statemachine Model Level j+1
Suite of Event-based Test scripts
Test Outcomes
trace
trace
trace
trace
Test analysis and reporting
Test analysis can navigate back up the traceability chain to report context of pass or fail
SUT running in Test Environment

Why Model Components & Interfaces?

- Modeling to the level of components is appropriate for defining and verifying architecture
- Interfaces and the services they offer are the external view of components that matters
- Why?
    1. IV&V arch verification should not mess with internals
    2. Model of component interfaces useful in verifying that components integrate as a working system
    3. Service concept; is there a provider for every required interface? Match of providers and consumers provides a static audit of completeness.

Why Model Interface Protocols?

1. Designing the interface is more than specifying the services one-at-a-time
   - Preconditions for successful invocation of a service are established by postcondition of a predecessor.
   - Successful maintenance of an invariant condition not to be disrupted by an intervening invocation.
2. Protocol Statemachines add dynamic view of how the services make a complete set
   - Audit of service preconditions and postconditions against the rules set out in the protocol establish a kind of dynamic completeness for the interface.

Why Model Interface …? Continued

3 Testing actual components is more than testing services one-at-a-time
   - Does the component reject illegal messages, even when syntactically well-formed, based on dynamic context?
   - Does the component respond to messages as specified by changes in what it will respond to?

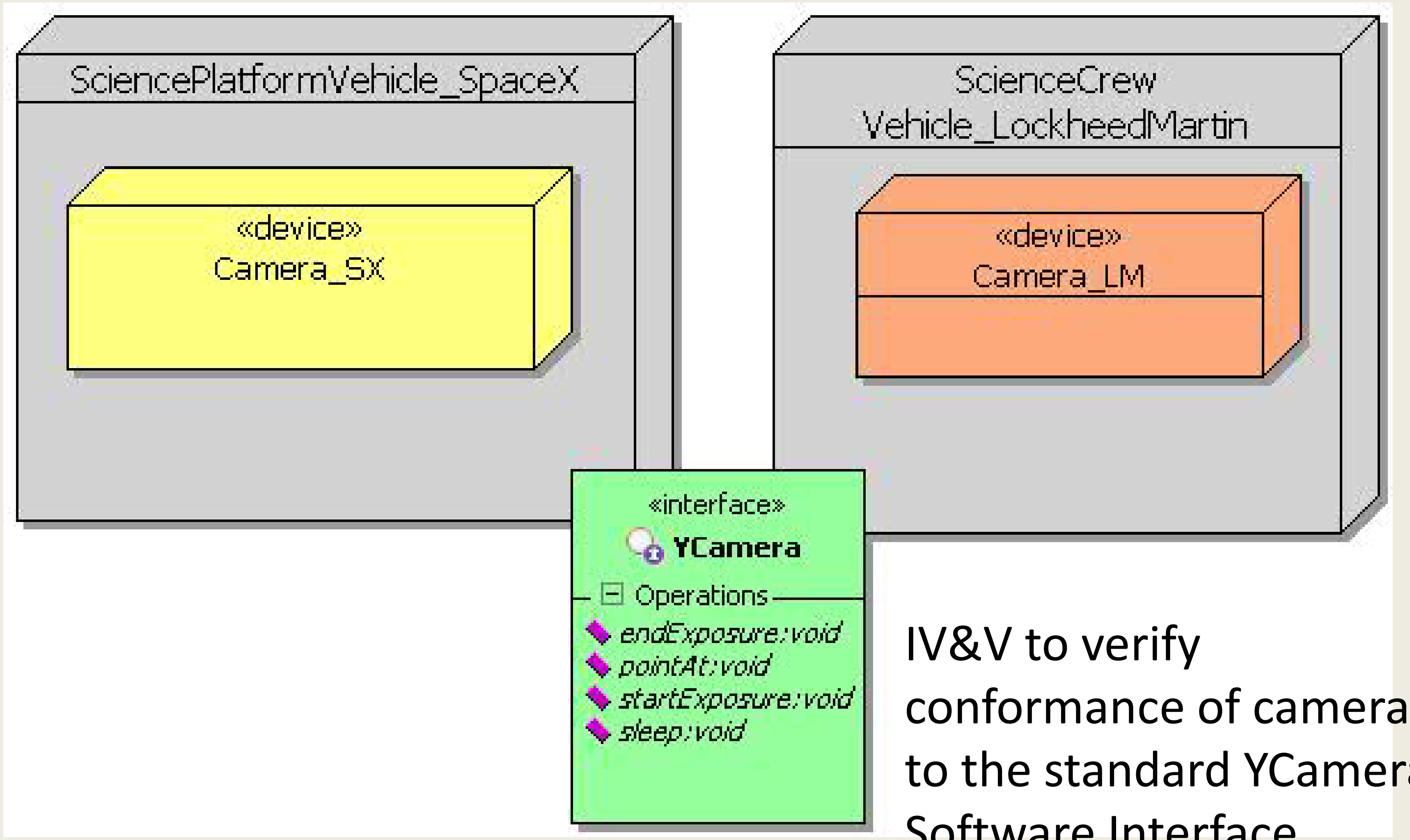
4 Actual behavior of implementations is mediated by protocols in Architecture Model, which thus support traces from implementation back to requirements
   - A behavioral statemachine and whitebox testing is not Architecture Verification – its place is later in cycle

Why Model Protocols .. Concluded

- Critically important for testing -- must be able to test actual components for conformance to published interfaces
  - Actual behaviors should conform to the protocol:
- Nominal case tests respect protocol: test driver who shifts without clutching has no right to complain of stripped gears.
- Status of tests that violate protocols is topic of debate, behavior of component SUT when service invocations violate protocol is undefined

Back to mission incident

Deployment Of Cameras from different vendors

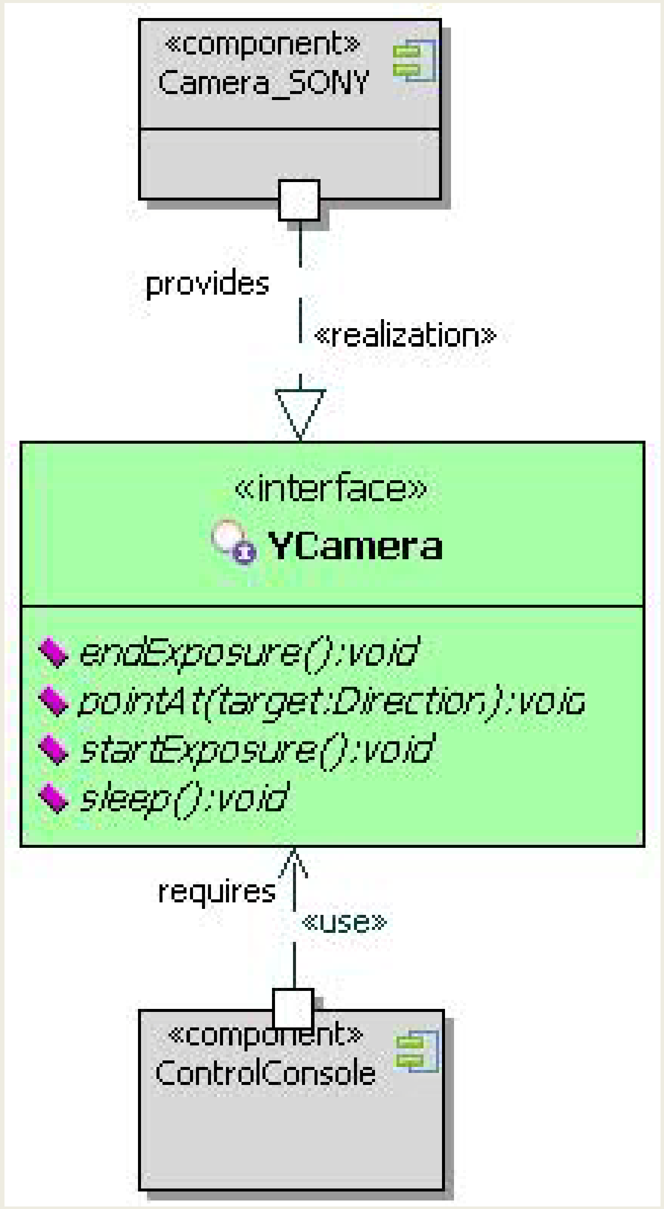


IV&V to verify conformance of cameras to the standard YCamera Software Interface

Y-Camera Commercialization

- Suppose Y-Cameras specified as components providing a SW interface
  - Y-Cameras can be provided by a number of qualified vendors
- Controlled by radiated commands originated at Mission Control, or by autonomous on-board software, or by other clients
- Y-Camera systems offer a software interface for functional control
- Any component implementing the Y-Camera specs can, on demand, *point at* a given environmental direction
  - Like a human cameraman told to point the camera in the direction of the actor starring in a scene, Cameras should track an assigned target, compensating for shifting platform attitude, until pointed elsewhere or deactivated.
- *Take-a-picture* function requires *start* and *end* exposure services
  - Can take a series of images while pointing at the same target
  - May want to point in the right direction first, await some event
- Deactivate device to conserve power, aka ***sleep***

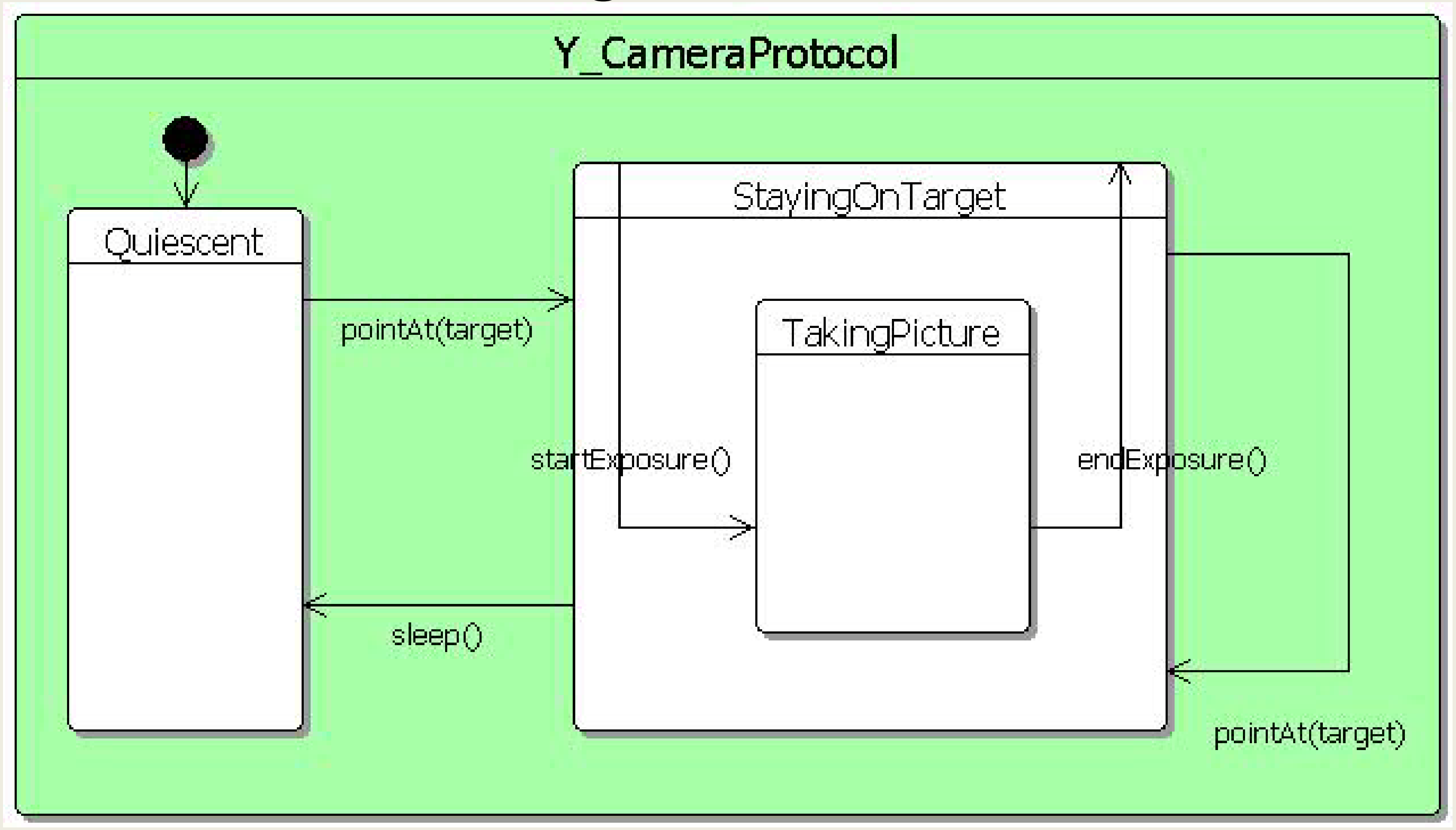
Four services exposed



- But they are not unrelated!
- How are they to be used?
- When you first establish communication with any Ycamera should you tell it to *startExposure*? Dynamic context, not static validation
- What state does it reach after 2 *pointAt* messages?
  - Note: not what will it DO
- Answers depend on the protocol statemachine

Protocol for using the interface

Contextually valid messages specified relative to context, meaning current state of interface



Tabular View

- Blue column labels show possible states
- Green row labels show possible events (messages or invocations)
- Black labels at crossings show next state, if protocol permits!
- Some messages are contrary to protocol in a given state
  - have no defined transition (non-deterministic)

	Quiescent	**StayingOnTarget**	**TakingPicture**
pointAt(target)	StayingOnTarget	StayingOnTarget	Not Legal
startExposure()	Not Legal	TakingPicture	Not Legal
endExposure()	Not Legal	Not Legal	StayingOnTarget
sleep()	Quiescent	Quiescent	Not Legal

Answers and discussion

What state reached by 2 pointAt(target) messages?

1. Depends on context

   current state, determined by the *prior* sequence of messages. Protocol says *pointAt(target)* should not be sent while camera is taking a picture. If this should happen, results undefined and likely to be undesirable.

2. This protocol statemachine does not address the question of whether the actual parameter (where to point) changes in successive *pointAt(target)* events.
3. Not the purpose of the protocol to model intended semantics of the *pointAt(target)* message as realized internally in the component

   Documentation on abstract method, aka service, *pointAt(target)*, owned by the YCamera interface, could show commonality among different realizations.

Summary

Architecture Verification using UML interface and protocol statemachine models

- Verification of software architectures by using UML(Unified Modeling Language) interface and statemachine models, in the context of the broader systems engineering problem of ensuring that complex systems can be integrated into a working whole.
- Introduction of the protocol statemachine concept using manual transmission example
- A failure this approach would have prevented: The case of the camera that did not take pictures because the command to take a picture needed to be preceded by a command to enable the camera.

Verification in General

- The goal of verification is, in general terms, proving that certain properties hold or do not hold, of some subject system. If the architecture is specified in terms of the UML and OO concepts, the composability of the system can only be established by using the concepts used by the architects.

OO style Software Architecture Verification in Particular

- More specifically, the property we are concerned with verifying is that the system can be produced by successful integration of a variety of separately produced subsystems, whose organization is described as in terms of components and interfaces as these are conceived in the OO paradigm. Hence, the topic of this presentation is *software architecture verification.*